

Metrix - Werkzeug zur Verwaltung von Messobjekten Wolfinger Reinhard

1) Die Aufgabe

Die Aufgabenstellung wurde der Übungsaufgabe 5 [1] aus Software Engineering am Institut für Wirtschaftsinformatik entnommen.

Das Messwerkzeug verwaltet Messobjekte. Messobjekte sind Verzeichnisse, Dateien (mit Quelltexten), Klassen und Methoden. Die Messobjekte stehen zueinander in verschiedenen Beziehungen:

- Eine Datei befindet sich in einem Verzeichnis.
- Ein Verzeichnis kann Dateien und/oder weitere Verzeichnisse enthalten.
- Jede Klasse ist in genau einer Datei definiert.
- Eine Datei kann Definitionen verschiedener Klassen enthalten.
- Zwischen Klassen können Vererbungsbeziehungen bestehen.
- Eine Klasse kann mehrere Methoden implementieren.
- Eine Datei samt Klassen ist einem Eigentümer zugeordnet.

Entwerfen und implementieren Sie eine objektorientierte Klassenbibliothek, die alle oben angeführten Messobjekte verwalten kann und einige einfache Statistiken liefert. Die Messobjekte haben Eigenschaften, wie z.B. Eigentümer (Entwickler), Programmversion und Anzahl der Quelltextzeilen (jeweils mit und ohne Kommentare). Das Programm soll im Einzelnen

- Messobjekte und deren Beziehungen zu anderen Messobjekten hinzufügen können.
- Zählmetriken für alle Messobjekte liefern können, z.B. Anzahl der Klassen und Anzahl der Methoden.

Die Klassenbibliothek soll leicht um neue Messobjekt-Typen erweitert werden können.

Entwickeln Sie ein Anwendungsprogramm, das die Funktionalität der Klassenbibliothek zeigt. Das Anwendungsprogramm kann eine textuelle oder grafische Benutzerschnittstelle haben.

2) Schnittstellen

Die Daten werden in Form einer Textdatei übernommen. Diese Textdatei wurde in Übungsaufgabe 4 mit einer attribuierten Grammatik (ATG) verarbeitet und geprüft. Diese Übungsaufgabe habe ich mit Coco implementiert. In dieser Aufgabe wird die syntaktische Korrektheit der Textdatei vorausgesetzt, weil sie bereits durch die ATG geprüft wurde.

```
PROGRAM: Mail
FILE: CLAEEngine.cs
OWNER: Markus Scharf
```

```

LANG: CSharp
DIR: S:\Sources\MEN\Implementation
CLASS: CEngine
BASE: CCollection, CServer, CObject
IMPLEMENTS: IEnumerable, IDisposable
METHOD: Foo(int num) 500 400 300
METHOD: Init() 50 40 30
METHOD: Copy() 160 30 20
CLASS: CMessage
BASE: CObject
METHOD: Copy() 310 120 40
FILE: CLAConverter.vb
OWNER: Michael Grund
LANG: VBasic
DIR: S:\Sources\FAD\Implementation
CLASS: CAnalyzer
BASE: CObject
METHOD: Join() 200 50 10
METHOD: Split() 450 90 60
METHOD: Copy() 55 15 5
PROGRAM.
    
```

Abbildung 1: Beispiel Textdatei Sample.metrix

3) Architektur

Das Klassenbibliothek wurde in einer Static Library CodeMetrics.lib implementiert. Die grafische Benutzeroberfläche Metrix.exe verwendet die Library CodeMetrics.

Als Grundlage für die Implementierung wurde die Entwurfsvariante B von Dr. Fröhlich aus [3] übernommen.

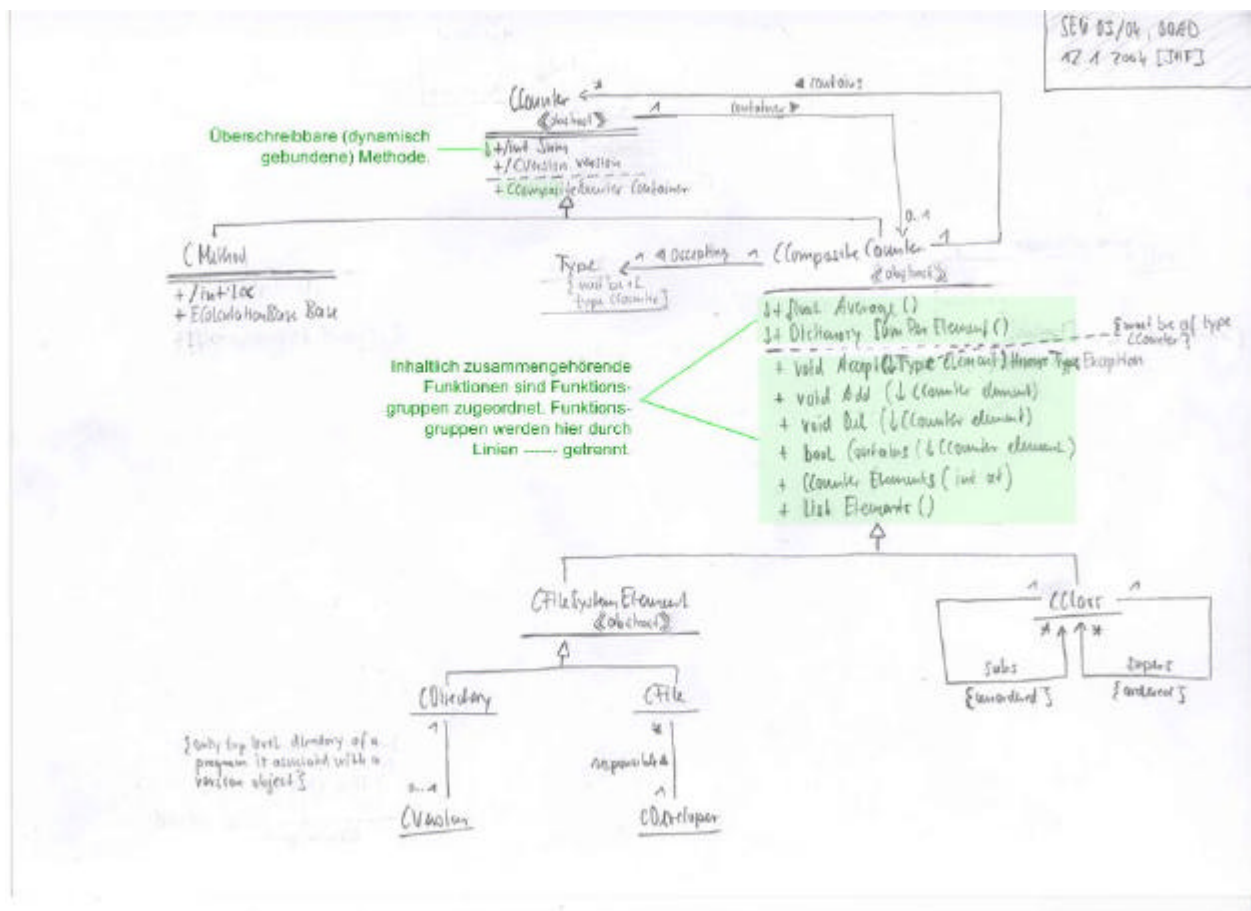


Abbildung 2: Entwurfsvariante B für Metrikenwerkzeug [3]

4) Benutzerhinweise für Metrix

Metrix wird mit dem Namen der Textdatei die angezeigt werden soll als Parameter aufgerufen.

```
METRIX [drive:][path]filename
```

Der Aufruf für die mitgelieferte Textdatei sample.metrix lautet wie folgt.

```
METRIX sample.metrix
```

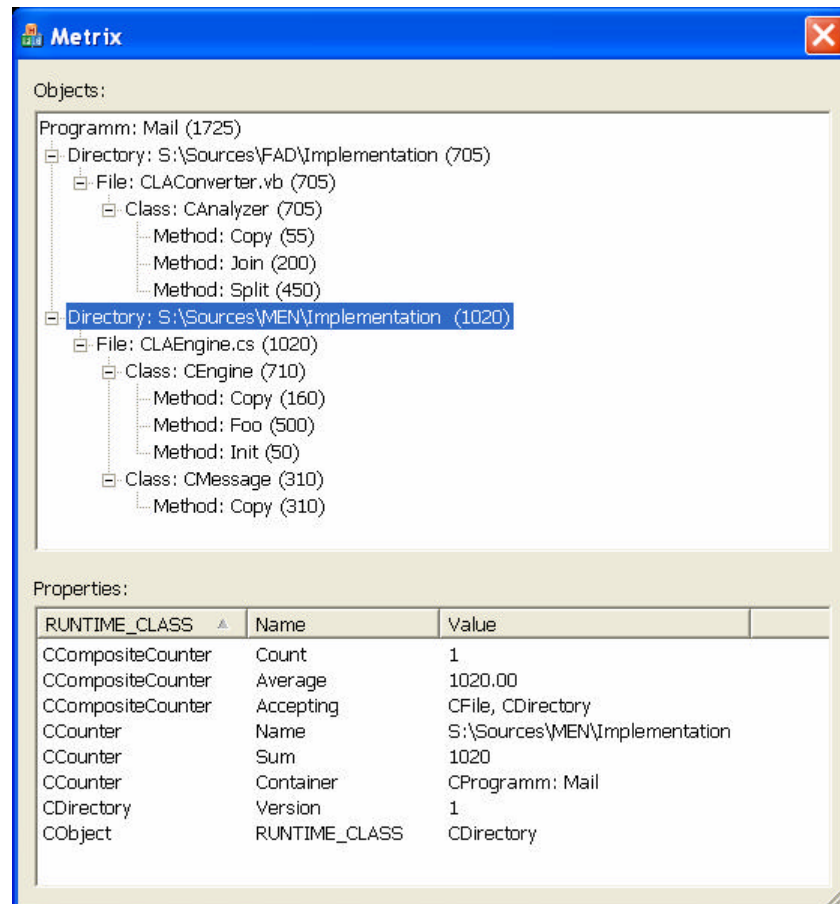


Abbildung 3: Benutzeroberfläche Metrix

Objects (CTreeCtrl)

Im oberen Fensterbereich wird die Datenstruktur hierarchisch dargestellt. Jedes Objekt wird durch folgende Daten dargestellt:

Typ: Name (Aggregierter Zähler)

- Typ: Typ des Objekts. Interessant ist hier, dass die Anzeige des Typs direkt aus dem Klassennamen erfolgt. Es wird einfach das führende "C" entfernt. So entsteht beispielsweise aus der Klasse CDirectory die Beschriftung "Directory".
- Name: Name des Objekts
- Aggregierte Zähler: In Klammern wird der für das Objekt aggregierte Zählstand angezeigt. In der Baumansicht ist gut zu erkennen, wie die Zähler der enthaltenen Objekte aggregiert wird.

Properties (CListCtrl)

Im unteren Fensterbereich werden für das bei "Objects" ausgewählte Objekt alle Eigenschaften angezeigt. Dabei wird unterschieden aus welcher Klasse der Klassenbibliothek die Eigenschaften kommen. Dabei gibt es Eigenschaften die alle Objekte gemeinsam haben, das sind jene aus der Klasse CCounter. Es gibt aber auch für die meisten Klasse, wie CDirectory oder CMethod, individuelle Eigenschaften.

- `RUNTIME_CLASS`: Die Klasse in der die Eigenschaft implementiert ist.
- `Name`: Name der Eigenschaft
- `Value`: Werte der Eigenschaft

Leistungsmerkmale der Benutzerschnittstelle

- Das Fenster kann in der Grösse verändert werden. Die Steuerelemente passen Ihre Grösse an (`CResizingDialog`). Die Fenstergrösse und -position werden in der Registrierung gespeichert.
- Bei der Listenansicht für die Eigenschaften können die Spalten umgeordnet und in der Breite verändert werden. Durch Klicken in den Spaltenkopf kann die Sortierung eingestellt werden. Die Spaltenanordnung und Sortiereinstellung werden in der Registrierung gespeichert (`CCharlesBronsonListCtrl`).

5) Implementierungstechniken

Im folgenden möchte ich einige interessante Implementierungstechniken und Prinzipien vorstellen.

Programmfluss über Exception gesteuert

Die Library CodeMetrics verwendet keine Funktionsrückgabewerte zur Fehlermeldung sondern setzt konsequent Exceptions ein. In der Benutzerschnittstelle können diese Exceptions an einer zentralen Stelle gefangen werden. Da alle Ausnahmen von `CExceptionExt` erben, kann über die Methode `GetErrorMessage()` eine Fehlermeldung mit der Ursache ausgegeben werden. Fehlermeldungen sind nicht in Prosa fomuliert ("Datei nicht gefunden") sondern liefern den enum-Wert von `CExceptionExt::GetCause()` als Zeichenkette ("`CImportException::DirectorySymbolMissing`"). Durch Einsatz diese Technik, bleibt der Codieraufwand für Fehlerbehandlung gering.

In der Benutzerschnittstelle gibt es nur ein einziges `CATCH`-Statement und keine sonstigen Abfragen eines Rückgabewerts im Stile von `if(!SUCCEDED())`. Das macht den Code gut lesbar.

```
class CExceptionExt : public CException
{
    DECLARE_DYNAMIC(CExceptionExt)
public:
    CExceptionExt(int cause);

    int GetCause() const;

    // this method is the one used by AfxProcessWndProcException()
    virtual BOOL GetErrorMessage(
        LPTSTR lpszError,
```

```

        UINT nMaxError,
        PUINT pnHelpContext = NULL);

    virtual LPCTSTR GetCauseAsString() = 0;

private:
    int m_cause;
};

```

Abbildung 4: Klasse CExceptionExt

Keine ungarische Notation

Metrix ist mein erstes CPP-Projekt, bei dem ich nicht mehr die ungarische Notation bei Variablennamen verwende. Früher hätte ich Variablen wie folgt deklariert.

```

CFoo** ppFoo;
int nCnt;
CString strName;
LPCTSTR lpszBuf;

```

Und wenn ich später irgendwann den Typ ändern will, muss ich alle Verwendungen mit Suchen & Ersetzen ändern. Das ging mir schrecklich auf die Nerven. Darum habe ich mich entschieden den Typ nicht mehr in den Namen zu prefixen. Visual Studio zeigt den Typ ohnehin an vielen Stellen an. Ich meine die Lesbarkeit wird dadurch eher besser als schlechter.

```

CFoo** fpp;
int cnt;
CString name;
LPCTSTR buf;

```

Das gilt für alle Standardprefixes, wie Pointer, String und die Basisdatentypen. Bei HWND, RECT oder ähnlichen Dingen verwende ich weiterhin Prefixe.

Kein Namespace

Ich wollte eigentlich die Static Library in einen Namespace CodeMetrics verpacken. Es gab nämlich einen Namenskonflikt zwischen meiner Klasse CFile und der MFC-Klasse CFile. Diesen wollte ich durch den Namespace lösen, Nur hat das nicht funktioniert. Die MFC implementiert RTTI nicht über die CPP-Sprachmöglichkeit, sondern über die Makros DECLARE_DYNAMIC(class) und IMPLEMENT_DYNAMIC(class, base_class). Und genau diese Markros wollen den Namen der Klasse in einen Methodennamen umsetzen. Wenn ich jetzt DECLARE_DYNAMIC(CodeMetrics::CFile) schreibe, kann er mit dem Klassennamen keine Funktion erstellen, weil Funktionsnamen keine Doppelpunkte enthalten dürfen. Kurz - die Makros sind nicht auf Namespaces vorbereitet. Damit war der Namespace ausgeschieden und ich musste mir mit einem #define helfen.

```
#define CFile __CFile__
```

Alle Header precompiled

Es gibt nichts Schlimmeres als während des Codierens ständig weitere Headerdateien irgendwo inkludieren zu müssen. Durch die wechselseitigen Abhängigkeiten wird das schnell eine mühsame Arbeit. Ich hab der Einfachheit halber alle Klassendefinitionen in einen Header CodeMetrics.h inkludiert und diesen

in stdafx.h aufgenommen. Vorteil ist, ich brauche auch in der Benutzeroberfläche nur diese eine Datei CodeMetrics.h zu inkludieren.

Nachteil: Jede Änderung in einer Schnittstelle erfordert eine Neuübersetzung des precompiled header. Aber das Projekt ist so klein, das spielt die Compilezeit noch keine Rolle.

Keine Implementierung im Header, keine inline Funktionen

Ich habe in den Header-Dateien keine Implementierungen gemacht. Das hat damit zu tun weil ich alle Header in stdafx.h einbinde. Dann hätten Änderungen in den Implementierung teilweise auch Neuübersetzung des PCH zur Folge.

Zu Beginn hab ich über Performance nachgedacht und wo möglich Funktionen als inline deklariert. Das hat aber Schwierigkeit mit dem Buildprozess gebracht. Da die Aufgabe trivial ist und Performance keine Rolle spielt, habe ich alle inline Direktiven entfernt.

Konsequente Verwendung von TCHAR

Wo möglich und sinnvoll habe ich CString verwendet. An manchen Stellen, beispielsweise bei der Parameterübergabe wird als Typ LPCTSTR verwendet. Oder die Klasse CImportLine verwendet zur Mustererkennung eine struct IMPORTLINE in der TCHAR* Zeichenketten verwendet werden. Bei den Stringmanipulationsfunktionen der CRT verwende ich immer die jeweilige TCHAR-Variante (Beispiele: _tcslen, _tcsncpy).

Nur virtual was virtual sein muss

Ich verwende virtual nur dort, wo ich bewußt die Zusicherung mache, dass diese Methoden überschrieben werden können. In der überschreibe ich diese Klassen dann auch selbst. Dass später irgendjemand den Code als Basis für eigene Erweiterungen nimmt und dann mehr virtual brauchen würde, habe ich nicht berücksichtigt.

Schlanke Schnittstellen

Zwischen privaten Methoden will ich schlanke Schnittstellen haben. Wo möglich übergebe ich die Daten nicht als Methodenparameter sondern halte die Daten in Member-Variablen. Damit steigt meine Motivation den Code in viele kleine Methoden zu trennen bei denen alleine durch den Namen klar ist was sie tun. Sprechende Methodennamen statt Kommentar. Grundsatz: statt zwei Blöcke in einer Methode die durch einen Kommentar eingeleitet werden, gleich zwei getrennte Methoden.

Nützliche Makros

Wenn ich eine Variable oder einen Parameter nicht benutze, dann kann für den Leser unklar sein ob ich auf die Variable vergessen habe, oder sie absichtlich nicht benutze. Solle Variable markiere ich durch UNUSED_ALWAYS(dontNeedYou). Damit verzichtet der Compiler auf eine Warnung und dem Leser ist klar was gemeint ist.

Typsichere Container

Die Containerklasse `CCompositeCounter` ist typsicher ausgeführt. Das heisst man, kann bspw. nicht Klassen in Methoden einfügen. Im Konstruktor kann man über die Methode `Accept(CRuntimeClass*)` einstellen, welche Container-Elemente akzeptiert werden.

```
CDirectory::CDirectory(LPCTSTR path)
{
    Accept(RUNTIME_CLASS(CFile));
    Accept(RUNTIME_CLASS(CDirectory));
}
```

CResizingDialog

Diese Klasse implementiert einen `CDialog` der in der Größe veränderbar ist. Dabei wird über einen struct eingestellt, wie sich die einzelnen Steuerelemente bei Größenänderungen verhalten sollen.

```
typedef struct tagADJUSTCTRLINFO
{
    UINT nID;
    bool fAdjustLeft;
    bool fAdjustTop;
    bool fAdjustWidth;
    bool fAdjustHeight;
}
ADJUSTCTRLINFO, *LPADJUSTCTRLINFO;
```

Abbildung 5: ADJUSTCTRLINFO (ResizingDialog.h)

```
static ADJUSTCTRLINFO adjustInfo[] = {
// ID          L      T      W      H
  { IDC_S_TREE,    false, false, true,  false },
  { IDC_TREE,     false, false, true,  true  },
  { IDC_S_LIST,   false, true,  true,  false },
  { IDC_LIST,     false, true,  true,  false }
};
```

Abbildung 6: adjustInfo (MetrixDlg.cpp)

Man kann sich das so vorstellen. Dort wo in der Struktur ein `false` steht ist die Ecke des Steuerelements quasi verankert. Dort wo ein `true` steht wie Position und/oder Größe angepasst. Die in der Ressource eingestellte Größe wird als Mindestgröße eingestellt, über die Nachricht `WM_GETMINMAXINFO` wird verhindert, dass der Dialog kleiner als diese Mindestgröße gemacht wird.

In der rechten unteren Fensterecke wird mit der Klasse `CGripperWnd` ein "Angreifer" optisch dargestellt. Damit kann man das Fenster auch durch Klick in den Gripper in der Größe verändert. Zusätzlich zum Klick in den Fensterrahmen.

CCharlesBronsonListCtrl

Diese Klasse unterstützt das Handling von `CListCtrl`'s wie die Darstellung der Eigenschaften im unteren Fensterbereich. Die eigentliche Stärke spielt das Steuerelement bei großen Datenmengen aus. Dann nämlich werden die Daten erst für den Inhalt erst dann nachgeladen wenn ein Eintrag sichtbar wird.

Für diese Anwendung nutzen wir aber einige einfachere Funktionen: Erstellen der Spalten, automatische Sortierung, Persistenz von Spaltenanordnung und Sortierung in der Registrierung.

Beispiel: Erstellung der Spalten. Man definiert in der Ressource-Datei folgende Stringressourcen.

```
IDS_DETAILS_NUMCOLS    "3"
IDS_DETAILS_COL1      "RUNTIME_CLASS~30"
IDS_DETAILS_COL2      "Name~30"
IDS_DETAILS_COL3      "Value~40"
```

Die Zahlen hinter der Tilde ~ geben den prozentualen Anteil einer Spalte an der gesamten Listenbreite dar. Die Erstellung der Spalten geht in einem Aufruf.

```
m_list.CreateColumns(IDS_DETAILS_NUMCOLS);
```

Über eine Struktur können die Sortierung und die Persistent aktiviert werden.

```
typedef struct tagCBLCINFO
{
    tagCBLCINFO();

    DWORD   style;
    DWORD   mask;
    LPTSTR  iniSec;
    LPTSTR  iniEntryFmt;
    int     sortAsc; // imagelist index for icon "sort ascending"
    int     sortDesc; // imagelist index for icon "sort descending"
    LPBOOL  lpbIgnoreItemChanged;
}
CBLCINFO, *LPCBLCINFO;
```

Abbildung 7: CBLCINFO (CBLListCtrl.h)

Metrix aktiviert die Funktionen durch den folgenden Code:

```
CBLCINFO lci;
::ZeroMemory(&lci, sizeof(lci));

lci.style = CBLCS_FULLROWSEL;
lci.mask = CBLCM_WIDTH | CBLCM_ORDER | CBLCM_SORT;
lci.iniSec = _T("Details");
lci.iniEntryFmt = _T("DetailsLV");
```

Und schon kann man die List sortieren und die Spalteneinstellungen speichern.

6) Lines Of Code

60 Files

147464 Bytes

Filename	Code	Comment	Empty	Total
s:\codemetrics\Class.cpp	13	28	18	59
s:\codemetrics\CompositeCounter.cpp	108	50	106	264
s:\codemetrics\CompositeCounterException.cpp	19	27	19	65
s:\codemetrics\counter.cpp	14	29	20	63
s:\codemetrics\CounterException.cpp	18	27	19	64
s:\codemetrics\Developer.cpp	8	26	11	45

s:\codemetrics\DeveloperList.cpp	21	27	24	72
s:\codemetrics\Dictionary.cpp	1	24	2	27
s:\codemetrics\Directory.cpp	11	26	13	50
s:\codemetrics\ExceptionExt.cpp	13	26	13	52
s:\codemetrics\File.cpp	15	29	21	65
s:\codemetrics\FileSystemElement.cpp	2	23	2	27
s:\codemetrics\Import.cpp	125	56	92	273
s:\codemetrics\ImportException.cpp	23	27	19	69
s:\codemetrics\ImportLine.cpp	59	34	51	144
s:\codemetrics\Method.cpp	10	27	14	51
s:\codemetrics\Programm.cpp	29	32	31	92
s:\codemetrics\stdafx.cpp	1	24	2	27
s:\codemetrics\TypeException.cpp	19	27	19	65
s:\codemetrics\vchelp.cpp	71	34	69	174
s:\codemetrics\Version.cpp	8	26	11	45
s:\codemetrics\VersionList.cpp	20	27	24	71
s:\codemetrics\Class.h	12	24	7	43
s:\codemetrics\CodeMetrics.h	31	26	4	61
s:\codemetrics\CompositeCounter.h	31	39	14	84
s:\codemetrics\CompositeCounterException.h	13	26	10	49
s:\codemetrics\counter.h	14	24	7	45
s:\codemetrics\CounterException.h	10	25	9	44
s:\codemetrics\Developer.h	13	25	9	47
s:\codemetrics\DeveloperList.h	10	24	5	39
s:\codemetrics\Dictionary.h	2	24	2	28
s:\codemetrics\Directory.h	9	24	6	39
s:\codemetrics\ExceptionExt.h	13	25	8	46
s:\codemetrics\File.h	13	24	7	44
s:\codemetrics\FileSystemElement.h	3	24	4	31
s:\codemetrics\Import.h	32	30	10	72
s:\codemetrics\ImportException.h	15	25	9	49
s:\codemetrics\ImportLine.h	35	24	11	70
s:\codemetrics\Method.h	14	24	10	48
s:\codemetrics\Programm.h	16	30	8	54
s:\codemetrics\stdafx.h	10	24	6	40
s:\codemetrics\TypeException.h	11	25	9	45
s:\codemetrics\vchelp.h	16	34	10	60
s:\codemetrics\Version.h	9	24	6	39
s:\codemetrics\VersionList.h	10	24	6	40
s:\metrix\CBLListCtrl.cpp	402	70	270	742

s:\metrix\CommandLineInfoEx.cpp	35	36	29	100
s:\metrix\GripperButton.cpp	42	28	27	97
s:\metrix\Metrix.cpp	36	38	37	111
s:\metrix\MetrixDlg.cpp	193	51	141	385
s:\metrix\ResizingDialog.cpp	84	35	54	173
s:\metrix\stdafx.cpp	1	23	1	25
s:\metrix\CBLListCtrl.h	64	37	30	131
s:\metrix\CommandLineInfoEx.h	9	24	6	39
s:\metrix\GripperButton.h	7	24	6	37
s:\metrix\Metrix.h	16	24	10	50
s:\metrix\MetrixDlg.h	30	28	14	72
s:\metrix\ResizingDialog.h	34	35	18	87
s:\metrix\resource.h	21	6	1	28
s:\metrix\stdafx.h	33	26	13	72
TOTAL	1957	1739	1434	5130
Total CodeMetrics.lib	950	1254	777	2.981
Total Metrix.exe	1007	485	657	2.149

7) Sourcecode - Workspace "Metrix"

Das Archiv metrix.zip muss in ein Wurzelverzeichnis ausgepackt werden. Der Laufwerksbuchstabe ist dabei egal.

- \Metrix.dsw (Workspace)
- \CodeMetrics\CodeMetrics.dsp (Projekt)
- \Metrix\Metrix.dsp (Projekt)
- \Sample.metrix (Testdatei)

Zum Build wird Visual Studio .NET 2003 (C++) benötigt. Zusätzlich ist eine Variante angepasst für Visual C++ 6.0 verfügbar.

8) Referenzen

- [1] SE-Übung 5: Verwaltung von Messobjekten;
http://www.se.jku.at/teaching/lva/ws03-04/se_uebung/03_angaben/uebung_5/uebung_5.pdf
- [2] SE-Übung 4: Verwaltung strukturierter Datenströme;
http://www.se.jku.at/teaching/lva/ws03-04/se_uebung/03_angaben/uebung_4/uebung_4.pdf
- [3] Entwürfe zum Metriktool; http://www.se.jku.at/teaching/lva/ws03-04/se_uebung/05_gruppen/froehlich/ood/index.html
- [4] LOCC - Lines of Code Counter; <http://centcom.sitefactory.at/locc.htm>